Proceedings of the 2005 IEEE
Engineering in Medicine and Biology 27th Annual Conference
Shanghai, China, September 1-4, 2005

GPU-based Volume Rendering for Medical Image Visualization

Yang Heng[1] and Lixu Gu[1]
[1]Shanghai Jiaotong University 1954 Huashan Road, Shanghai, 200240, P.R.China
E-mail:yhq@sjtu.edu.cn

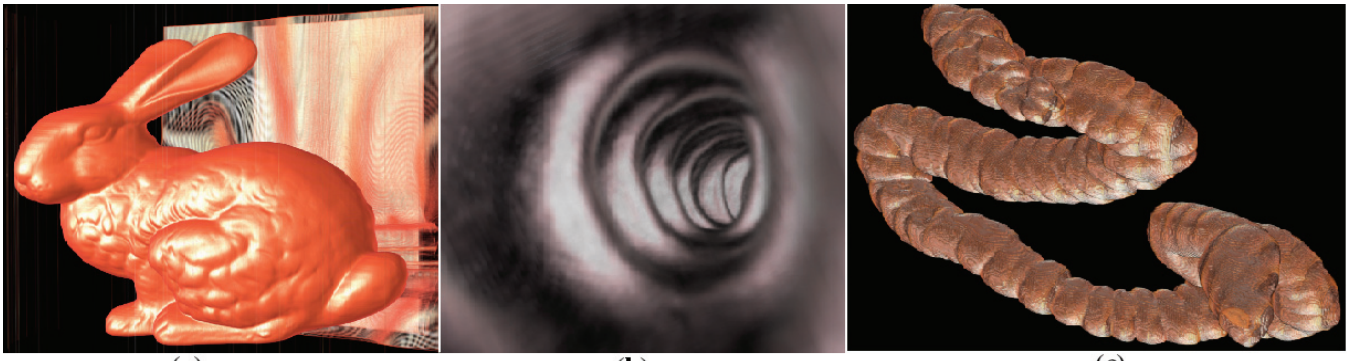

Figure1. The examples of volume rendering results using proposed approach: (a) CT data set from the Stanford terra-cotta bunny; (b) and (c) the segmented intestine from a CT clinic data employed in our Virtual Colonoscopy System

Abstract— **During the quick advancements of medical image visualization and augmented virtual reality application, the low performance of the volume rendering algorithm is still a "bottle neck". To facilitate the usage of well developed hardware resource, a novel graphics processing unit (GPU)-based volume ray-casting algorithm is proposed in this paper. Running on a normal PC, it performs an interactive rate while keeping the same image quality as the traditional volume rendering algorithm does.**

Recently, GPU-accelerated direct volume rendering has positioned itself as an efficient tool for the display and visual analysis of volume data. However, for large sized medical image data, it often shows low efficiency for too large memories requested. Furthermore, it always holds a drawback of writing color buffers multi-times per frame. The proposed algorithm improves the situation by implementing ray casting operation completely in GPU. It needs only one slice plane from CPU and one 3Dtexture to store data when GPU calculates the two terminals of the ray and carries out the color blending operation in its pixel programs. So both the rendering speed and the memories consumed are improved, and the algorithm can deal most medical image data on normal PCs in the interactive speed.

Keywords— **GPU acceleration, direct volume rendering, medical image data, 3Dtexture**

I. INTRODUCTION

Direct volume rendering is a powerful tool in the display and visual analysis of medical image data. Different to the common used surface rendering which could miss the most important part of the data by selecting the wrong iso-value, direct volume rendering constructs images in which all volume cells (voxel) can make a contribution to the final visualized results. So it can more efficiently display the inner structure of the objects than the surface rendering. But owing to the slow speed limitation, CPU-based volume rendering is seldom used on common PC.

With the rapid development of graphic card hardware, GPU-based volume rendering techniques show their great powers in accelerating rendering. The basic work involved is to store volume data in texture, resample and interpolate them using hardware instead of software. This idea was first introduced by Cullip and Neumann in 1994 [1], and further extended to advanced medical imaging by Cabral et al. in 1994[2]. Cabral et al. demonstrated that both interactive volume reconstruction and interactive volume rendering was possible with hardware providing 3D texture acceleration. Later, Rudiger Westermann et al. [3] introduced an accelerated volume Ray-Casting using 2D texture to store the ray terminals in 2001. Although its great improvement in producing an intermediate image, it still relies on CPU to cast the ray for every pixel. Further development of GPU-based ray-casting was described by Timothy J. Purcell et al. [4]. They stored a data link as 2D texture to calculate the interactions for ray casting in GPU programs. In 2003, a new ray casting algorithm was proposed by J. Kruger and R. Westermann [5]. They described a stream model on graphics hardware that is programmable via the Pixel Shader 2.0 API and implemented the standard acceleration techniques for volume ray-casting. Although it achieved the high speed, it consumes too much memory by its three 2D textures and one 3D texture. Still many others use GPU to carry out volume rendering (Allen Van Gelder and Kwansik Kim 1996[6]; Tae-Young Kim, Yeong Gil Shin 2001[7]; Wei Li et al. 2003[8]; Daniel Weiskopf et al. 2004 [9]).

Thought these algorithms hold great benefits, one drawback is often omitted, which is to write to color buffers several times per frame. On the contrary, both the ray terminals and the color blending operation are calculated in pixel programs in the proposed algorithm, so that it writes color buffer only once to further accelerate the volume rendering.

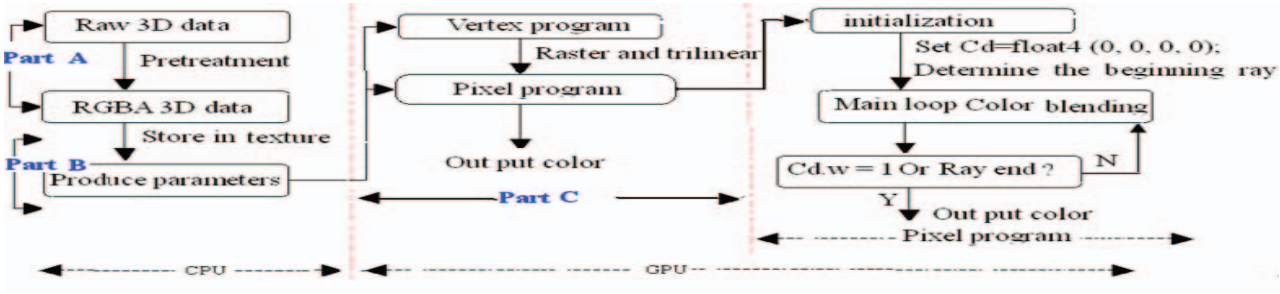

Fig.2. the data pipeline of the proposed algorithm

The paper is outspreaded in three parts. Part one explains the basic idea of the sliced-based 3Dtexture volume rendering. Part two describes the algorithm in detail, and Part three shows the test results and describes our discussion.

II. GPU-BASED VOLUME RENDERING

2.1 The traditional slice-based 3D texture volume rendering

The traditional slice-based 3Dtexture volume rendering is usually performed by slicing the volume in back-to-front or front-to-back order with planes oriented parallel to the view plane. For each fragment, program gets the sampled color from texture by trilinear interpolation and then blends it with the current value in color buffers by proper blending functions, which is described in equation 1.1~1.2.

$$C_d = (1 - A_s)C_d + A_s C_s \quad \text{.....(1.1)}$$

$$C_d = (1 - A_s)C_d + A_d C_s, A_d = A_d + (1 - A_d)A_s \quad \text{.....(1.2)}$$

Here, C_d A_d and C_s, A_s are the color and opacity values of the color buffer and the incoming fragment, respectively. For front-to-back order, the accumulated opacity will be stored in α-buffer and the α-test should be open.

The slice-based rendering describes a general process for GPU accelerated volume rendering. The proposed algorithm also uses 3Dtexture to store data and resample them in pixel processing. But in order to reduce computing time, it does not slice the cubes in CPU but casts ray in GPU. CPU only produces a polygon parallel to the view plane as the basic ray and prepares some necessary parameters for GPU.

2.2 Ray casting volume rendering on GPU

For many years, with the advancement of commodity graphics hardware and the creation of its new feature of programmability, many algorithms began to transfer some of the processing stages from CPU to GPU on common PCs. The reason is that with its intrinsic parallelism and efficient communication, GPU can calculate much faster than CPU. Furthermore, the power of GPU is currently increasing much faster than that of CPU's, and the algorithm realized on GPU can have great potentials in the future. Our algorithm takes advantage of NV40 GPU to implement volume rendering. NV40 GPU holds some new characteristics such as reading texture data in vertex programs, supporting dynamic shift command in pixel programs and even longer lengths for shader programs. These characteristics are the necessary factors in our algorithm. We use CG [10] to code GPU programs and C++, OpenGL and VTK to organize the data pipeline in CPU.

As shown in Fig 2, the algorithm is organized in three parts. Part A is to classify the raw data and add lightings. Part B discusses the parameters produced by CPU. Part C is the core of the algorithm and contains the whole ray casting progress.

2.3 The mathematic principle of the algorithm

Define 1 volume can be denoted by a cube in a 3D Descartes coordinate system, whose three borders are parallel to the axes respectively. The left-bottom coordinate is V_x,V_y,V_z. The cell number in three axes direction is X, Y, Z and the sampled distance in three directions is S_x,S_y,S_z respectively.

Lemma 1 The 3Dtexture coordinates T_{cord} of the point in Descartes coordinate system is respectively the linear function of the point's Descartes coordinates in three axes(equation 2.1~2.3). The parameters is defined in Define 1.and x, y, z are the world coordinate.

$$T_{cordx} = \frac{(x - V_x)}{S_x X} \quad \text{.....(2.1)}$$

$$T_{cordy} = \frac{(y - V_y)}{S_y Y} \quad \text{.....(2.2)}$$

$$T_{cordz} = \frac{(z - V_z)}{S_z Z} \quad \text{.....(2.3)}$$

Theorem 1 The ray casting function in Descartes coordinate system $R(t) = B + I \times t (0 <= t <= L)$ has its equivalence in 3Dtextue coordinates system. Here, B is the beginning ray position and I is the forward ray step when L stands for the total step number and t is the current step. The equivalences are shown in equation 3.1~3.3. The parameters are defined in **Define 1**.

$$TR_x(t) = \frac{I_x}{S_x \times X} \times t + \frac{B_x - V_x}{S_x \times X} \quad \text{.....(3.1)}$$

$$TR_y(t) = \frac{I_y}{S_y \times Y} \times t + \frac{B_y - V_y}{S_y \times Y} \quad \text{.....(3.2)}$$

$$TR_z(t) = \frac{I_z}{S_z \times Z} \times t + \frac{B_z - V_z}{S_z \times Z} \quad \text{.....(3.3)}$$

Theorem 2. The cube of the volume defined in Define 1 has a projection in the view plane and we can also find a rectangle just to include all of the projection points.

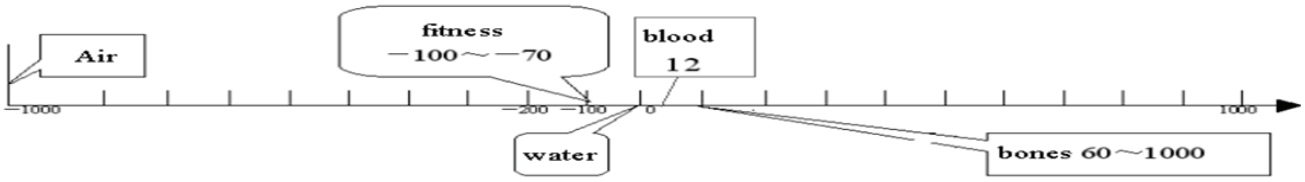

Figure 3. The CT value and its corresponding organ

Theorems 2 prove: First, project the eight corners of the cube to the view plane. And then transform the projected points to the screen coordinate and get the minimum and maximum coordinates in X, Y axes: X_0, X_1, Y_0,Y_1. Lastly, translate the four vertexes (X_0, Y_0, Z), (X_1, Y_0, Z), (X_0, Y_1, Z), (X_1, Y_1, Z) back to the world coordinate to get the rectangle. Here Z is the depth of the projection plane.

2.4 Preprocess raw data in CPU

As shown in Fig.3, the raw value of each voxel in a medical data (such as CT, MRI) represents the gray scale attribution of a particular organ. In order to emphasis the ROI, the algorithm needs to classify all the voxels to color values which can be thought as the colors that voxel absorbs or emits. So assign zero opacity to hide that voxel and a positive opacity to display it. Based on this principle, the proposed algorithm classifies the voxles by assigning RGBA values to the typical organ points and the others get color values by linear interpolate.

After classification, store RGBA colors to 3Dtexture and set the texture sampling interpolation style to be bilinear and the clipping coordinate style to no clipping.

2.5 Produce parameters in CPU

In order to implement ray casting operation in pixel program, CPU needs to prepare some parameters. Firstly, it requires a slice plane as the basic ray for ray casting operation. Use the sight direction as the normal and the nearest point and the farthest point in the volume cube to build up two planes as P_{near} and P_{far}. For front-to-back order rendering, the projection plane is P_{near} and for back-to-front order rendering P_{far}. Then define the projection rectangle by the method introduced in Theorem 2 and calculate the four vertexes' world coordinates V0 and texture coordinates T0 using Lemma 1. Secondly, calculate the ray casting function. Take the parallel projection as example; the function in world coordinates is equation 4.

$$R(t) = V_0 + I \times t (0 <= t <= L) \ldots\ldots (4)$$

Here V_0 is the beginning ray position, I is the increment of one step and L is the total step. For front-to-back order rendering, R(0) is in the P_{near} and R(L) is in the P_{far}. For back-to-front order R(L) is in the P_{near} and R(0) is in the P_{far}. According to **Theorem 1**, the corresponding ray casting functions in texture coordinates can be calculated using equation5.

$$T(t) = T_0 + \Delta T \times t (0 <= t <= L) \ldots\ldots (5)$$

Here t stands for the current step of the ray. ΔT is the increment of one step and L is the total step.

The last parameter is the texture size S_t. Since texture coordinates calculated by Lemma 1 may be out of range (0, S_t), St is designed to prevent the redundant parts in the ray. Finally, the algorithm transfer ΔT, L, S_t as uniform parameters to GPU and transfer V_0 and T_0 as binding parameters by drawing the cube using the four vertexes of the rectangle.

2.6 Ray casting in GPU program

As shown in the data pipeline in Fig 2, the third part runs on GPU in pixel programs, where two-step procedure is designed to carry out the ray casting. One is to initiate the ray terminals and another is to cast ray to calculate colors for that pixel.

Firstly, determine two terminals of the ray for each pixel. According to Theorem 1, the texture coordinates of the point inside the volume cube lies in the range of (0, S_t) while the point outside lies outside the range of (0, S_t). But for every pixel, the initial point position in the projection plane may be out of the cube so the texture coordinate is out of the range of (0, S_t). In order to reduce the ray steps, the algorithm needs to find the actual two terminals of the ray. For its linear character, the ray holds its redundant parts just in the two terminals. So By getting rid of the redundancy, the algorithm finds the beginning ray and the ending ray position. The beginning ray can be achieved by setting a proper t=L1 to make the texture coordinate just in the range (0, S_t) while the ending ray by setting a proper t=L2 to let texture coordinate just out of the range (0, S_t). Thus the beginning ray position (T_{begin}), the ending ray position (T_{end}) and the new casting function can be gotten. Secondly, cast the ray from Tbegin to Tend, where equation 1.1 is used for back-to-front order rendering and equation 1.2 for front-to-back order rendering. In front-to-back order, the ray can be ceased immediately when opacity equal to 1. After the casting process, the program outputs the pixel color.

III. EXPERIMENT

In order to test the effectiveness of the algorithm, we employed four datasets in our experiment. Dataset 1 is a CT scan of the Stanford terra-cotta bunny which as a size of 512*512*360; dataset 2 is a real clinical human abdomen CT data, which is in size of 400*400*344; dataset 3 and dataset 4 are two MRI head scans, which are in size of 190*217*190 and 256*256*109, respectively. Dataset 1 and 2 are downloaded from the Stanford volume data archive[1].

The experiment employs a NVIDIA NV40 GPU and is programmed under a Visual studio 6.0 environment on a Windows XP operation, where the GPU programs are coded with CG [10] language. The OpenGL and VTK libraries are used in the software and the rendering is directed to a 512*512 view port.

[1] http://www.graphics.stanford.edu/data/voldata/

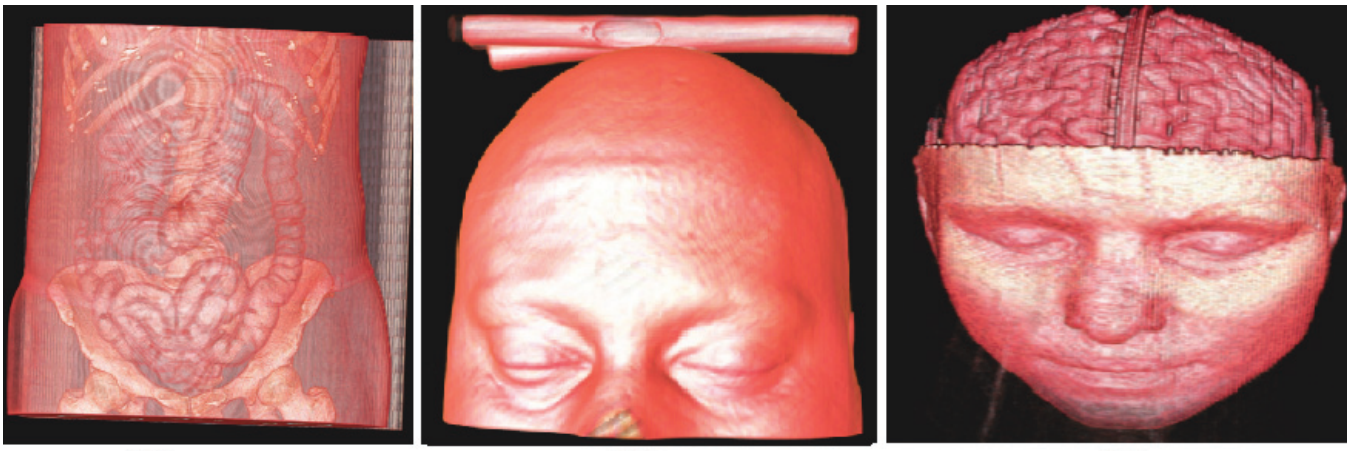

Figure 4. The examples of the results produced by proposed method. (a) CT abdominal scan (b) MR brain and (c) MR study of head with skull partially removed to reveal brain

The compare of the speed is described in table 1. The results of dataset 1 and the segmented intestine part from dataset 2 by the proposed algorithm are shown in Fig.1. Fig 4 shows the dataset 2 and the two MRI heads dataset 3 and 4.

In real medical application, the algorithm can employ GPU and CPU in parallel, so that it can realize an interactive speed on common PC while keeping the same image quality as the traditional rendering. With the rapid development of GPU, this algorithm will even show greater power in the future. As shown in Fig1, The rendering results (b) and (c) is two snapshots of a real Virtual Colonoscopy system where the proposed real time volume rendering has been achieved in a normal PC.

Table 1 Speed (fps) comparison between the three algorithms

	Dataset 1	Dataset 2	Dataset 3	Dataset 4
Ray Casting	0.40	0.37	1.29	0.50
3DTexture	1.38	2.06	3.19	3.4
GPU RayCasting	10.75	12.45	24.93	19.91

IV. CONCLUSIONS

This paper presented a novel volume-rendering algorithm for medical image visualization using NV40 GPU. Based on the flexible programming model of FV40 pixel shader, we build up a new data pipeline to implement ray-casting operation completely in GPU. Different from most other GPU-based volume rendering, this algorithm writes frame buffer only once and used the mathematic calculation determining the ray terminals to avoid the redundant texture storing them. So both time and the memories are reduced.

When classifying the raw volume rendering, the algorithm employed a pre-classification method to classify voxels before interpolation. Since this must be done in CPU, it still costs much time. The future work will address this problem by moving classification operation into GPU to get even quicker interactive speed.

V. ACKNOWLEDGEMENT

This work is partly supported by NDI

REFERENCES

[1] Timothy J. Cullip and Ulrich Neumann, “Accelerating volume reconstruction with 3D texture hardware”, Tech. Rep. TR93-027, University of North Carolina, Chapel Hill NC, USA, 1994.

[2] Brain Cabral, Nancy Cam and Jim Foran,“Accelerated volume rendering and tomographic reconstruction using texture mapping hardware”, Symposium on Volume Visualization, ACM Press New York, NY, USA, pp.91–98, 1994.

[3] Rudiger Westermann and Bernd Sevenich, “Accelerated Volume Ray-Casting using Texture Mapping”, 12th IEEE Visualization, pp.271-218, 2001.

[4] Timothy J. Purcell, Ian Buck, William R. Mark, and Pat Hanrahan. “Ray Tracing on Programmable Graphics Hardware[J]”. ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH), Volume 21, Issue 3, pp.703-712, 2002

[5] J. Kriiger and R. Westermann, “Acceleration Techniques for GPU-based Volume Rendering” , 14th IEEE Visualization, Seattle, Washington, USA, pp. 287-292, 2003.

[6] Allen Van Gelder, Kwansik Kim, “Direct Volume Rendering with Shading via Three Dimensional Textures”, IEEE Proceedings of the 1996 symposium on Volume visualization, San Francisco, California, USA, pp..23-30, 1996

[7] Tae-Young Kim and Yeong Gil Shin, “Fast volume rendering with interactive classification”, Computers & Graphics25, pp.819-831,2001.

[8] Wei Li, Klaus Mueller, and Arie Kaufman, “Empty Space Skipping and Occlusion Clipping for Texture-based Volume Rendering”, 14th IEEE Visualization, Seattle, Washington, USA, pp.317-324, 2003.

[9] Daniel Weiskopf, Manfred Weiler and Thomas Ertl, “Maintaining Constant Frame Rates in 3D Texture-Based Volume Rendering”, IEEE Proceedings of the Computer Graphics International (CGI’04), pp. 604-607, 2004.

[10] William R. Mark, R. Steven Glanville, Kurt Akeley and Mark J. Kilgard, “Cg: A system for programming graphics hardware in a C-like language”, Proceedings of SIGGRAPH, 2003.